Fast Newton-Raphson Power Flow Analysis Based on Sparse Techniques and Parallel Processing

Afshin Ahmadi ⬤, *Member, IEEE*, Melissa C. Smith ⬤, *Senior Member, IEEE*,
Edward R. Collins, *Senior Member, IEEE*, Vahid Dargahi ⬤, *Member, IEEE*,
and Shuangshuang Jin, *Senior Member, IEEE*

Abstract—**Power flow (PF) calculation provides the basis for the steady-state power system analysis and is the backbone of many power system applications ranging from operations to planning. The calculated voltage and power values by PF are essential to determining the system condition and ensuring the security and stability of the grid. The emergence of multicore processors provides an opportunity to accelerate the speed of PF computation and, consequently, improve the performance of applications that run PF within their processes. This paper introduces a fast Newton-Raphson power flow implementation on multicore CPUs by combining sparse matrix techniques, mathematical methods, and parallel processing. Experimental results validate the effectiveness of our approach by finding the power flow solution of a synthetic U.S. grid test case with 82,000 buses in just 1.8 seconds.**

Index Terms—**Parallel, multicore, sparse, power flow, newton-raphson, OpenMP, SIMD.**

I. INTRODUCTION

POWER systems throughout the world are undergoing a significant transformation, mainly driven by the increase in penetration of renewable energy, integration of distributed energy resources, and advances in digital technologies. Future grids are powered by clean energy, enable the bidirectional flow of electricity, and have a centralized-decentralized control scheme. The resulting benefits are improved reliability and resiliency, more efficient supply and delivery of energy, reduced environmental impact, and cost-effective energy generation. However, this transformation will make the planning, operation, and control of the power grid more complicated and computationally demanding. The variability and uncertainty of decision variables result in complex models that must be solved

more frequently and in a shorter time scale to handle the system dynamics. As a result, it is necessary to develop faster mathematical methods and utilize more computational power to ensure the adequacy and security of energy delivery. Fortunately, advancements in computer technology and the widespread availability of multicore processors open up new possibilities to address this problem.

Parallel computing has proven as a viable solution to accelerate several computationally intensive applications in the power system [1]–[3]. These applications typically belong to real-time control and simulation, optimization, and probabilistic assessment. Many of these studies rely on the solution of power flow (PF) within their process. PF analysis aims to obtain voltage magnitudes and angles at load buses, real and reactive power flow through the transmission lines, and voltage angles and reactive power injection at generator buses. This information is essential to determine the steady-state condition of the network and ensure the security and stability of the grid. However, PF is a non-linear, computationally demanding problem, where the solution is only meaningful for a short time since the state of the power system continuously changes.

Over the years, researchers have developed various numerical and analytical methods to solve power flow, such as Newton-Raphson (N-R), Gauss-Seidel (G-S), Fast Decouple (FD), Holomorphic Embedding, Continuation, and several others. To date, Newton-Raphson is still the most commonly used method because of its quadratic convergence property. On the other hand, this method requires more computational resources compared with others [4]. Researchers around the world have investigated the possibility of using parallel processing to boost the Newton-Raphson Power Flow (NRPF) performance. These attempts can be classified based on the parallel programming model and processor architecture: shared memory, distributed memory, graphic processing unit (GPU), configurable chips, and hybrid approaches. Only a few studies have employed the shared and distributed memory models to accelerate the solution of the NRPF algorithm on multicore processors. In [5], a maximum speedup of $5.6\times$ was achieved for a 1,944-bus system by employing OpenMP and eight CPU cores. Reference [6] reported a speedup of $3.3\times$ using four threads and OpenMP for a power system with 1,354 buses. Guera and Martinez-Velasco [7] integrated the high-performance Intel MKL PARDISO solver into the source code of the MATPOWER [8] library and the solution of a 9,241-bus test system was found in 560 ms.

Distributed computing for the solution of NRPF was investigated in [9] and [10], with an average speedup of $2\times$ in both studies. The communication time delay between nodes limits the performance of the distributed model. Additionally, authors in [11] examined the possibility of solving PF equations on field-programmable gate arrays (FPGAs) and achieved about 6x higher performance compared with the CPU implementation. However, using FPGA devices for accelerating power system applications has not received much attention by engineers and scientists since it requires significant programming effort and expertise.

Over the past decade, there have been enormous advancements in GPU hardware and programming tools, which has made these devices more powerful, popular, and accessible. As a result, a significant number of researchers have focused on GPU-based implementations of both PF algorithms and applications that are reliant on PF calculation such as contingency analysis [12]–[15]. A hybrid GPU-CPU NRPF implementation based on vectorization parallelization and sparse techniques was proposed in [16], where the solution of a 3,012-bus system was calculated in 206 ms. Although the results are promising, unfortunately, the scalability and effectiveness of the proposed approach is unknown since test systems are limited to 3,012-bus and less. A GPU version of three different PF algorithms was developed in [17] using the CUDA library and dense matrices. For a system with 974 buses, the run time was 19.6, 10.8, and 5.5 seconds for G-S, N-R, and P-Q decoupled power flow methods, respectively. Moreover, the performance of Gauss-Jacobi and N-R power flow algorithms on GPUs was studied in [18]. Results for a 4,766 bus system shows an execution time of 3.98 seconds for N-R and 3.06 seconds for Gauss-Jacobi. Furthermore, authors in [19] have compared the parallel implementation of N-R and G-S algorithms on both CPUs and GPUs using sparse techniques with the objective of accelerating the concurrent PF calculation of many cases of a single network. Reported speedup for a 2,383-bus system ranges from $6\times$ to $13\times$ depending on the target architecture and number of simultaneous runs. In the case of the G-S power flow method, although the reported speedups in [17]–[21] are better than the sequential G-S algorithm, this PF method cannot compete with the NRPF method since it is prone to divergence and requires many iterations to obtain a solution. Moreover, the nonlinear devices in the network cannot be modeled in the G-S method.

Additionally, references [22] and [23] sought to execute the fast decoupled power flow (FDPF) on GPUs. In [22], exploiting both a GPU-based preconditioned conjugate gradient solver and an inexact Newton method improved the performance up to 2.86 times for test systems upto 13,173 buses. Researchers in [23] compared the performance of running FDPF on GPUs with two different fill-in reduction algorithms, namely, reverse Cuthill-Mckee (RCM) and approximate minimum degree (AMD). By utilizing CUDA libraries, sparse matrices, and the AMD algorithm, a 4.19x speedup was achieved for a 13,659 bus power system. Parallel implementation of the PF algorithms for an unbalanced distribution network has been investigated in several studies as well [25]–[28]. However, this topic is beyond the

scope of this study since our focus is on balanced transmission networks.

Review of the literature shows that recent studies for accelerating PF calculations are mostly designed to exploit the computational power of GPUs. Although the reported speedups are encouraging, there are some disadvantages to this approach. First, the overhead time for data transfer between the host computer and GPU poses an obstacle for the performance of these implementations. Second, only a limited number of GPU-based sparse direct solver libraries are currently available and they typically utilize the entire device to find the solution of a linear system, which poses a significant limitation when it comes to applications that are reliant on concurrent execution of many PF scenarios. Finally, GPUs are generally equipped with less memory compared with shared-memory computers, which makes them undesirable for memory-intensive applications such as contingency analysis.

II. CONTRIBUTIONS

This paper aims to maximize the performance of Newton-Raphson power flow on multicore CPUs by combining software techniques, mathematical methods, and parallel processing. The main contributions of this study are summarized as follows:

- Both SIMD (Single Instruction, Multiple Data) vectorization and multicore processing are targeted to accelerate power flow calculations.
- The implementation employs the Compressed Sparse Row (CSR) storage format to address two major constraints in computing PF solution - memory and time.
- A parallel approach for forming and updating the sparse Jacobian arrays is introduced that significantly improves the execution time.
- The nested dissection algorithm is applied to reduce the *fill-in* and improve the computation time and memory usage in solving the system $Ax = b$.
- Various combinations of scenarios are benchmarked on power systems ranging from 1,354 to 82,000 buses.
- The proposed NRPF implementation is combined with contingency analysis, and results are presented to further demonstrate the significance of this research.

III. POWER FLOW PROBLEM

The PF study aims to calculate the bus voltages and power flow in the network given the nodal admittance matrix (Y_{bus}), known complex power at load buses (PQ), known voltage magnitude and angle for the slack generator bus, and known voltage magnitude and injected real power for the remaining generator buses (PV). Network equations in a power system are commonly formulated by the node-voltage method, which results in a complex linear system of equations in terms of injected bus currents,

$$I = Y_{bus}V \tag{1}$$

However, the complex power values are generally known in a power system rather than the bus currents. Thus, Eq. (1) should

TABLE I
PSEUDO-CODE OF IMPLEMENTED NEWTON-RAPHSON POWER FLOW

	Function	Comment
1	**Build Ybus**	form admittance matrix
2	**Power Flow Loop**	
3	memory allocation	one-time run
4	initial calculations	one-time run
5	calculate partial derivatives	performed at every iteration
6	form Jacobian mapping arrays	one-time run, sparse only
7	update Jacobian matrix	performed at every iteration
8	solve the linear system	performed at every iteration
9	update voltages	performed at every iteration
10	calculate power mismatch	performed at every iteration
11	calculate norm	performed at every iteration
12	**Power Flow Solution**	
13	update generators P and Q	active and reactive
14	calculate branch flows	active and reactive
15	calculate line losses	active and reactive
16	calculate line charging injections	required for OPF
17	generate misc. information	violations, etc.

be modified to integrate complex power equations, which results in a set of nonlinear power flow equations. The injected complex power into the ith bus is,

$$S_i = P_i + jQ_i = V_i I_i^* = V_i \left(\sum_{k=1}^{n} Y_{ik}^* V_k^* \right) \quad (2)$$

where P_i and Q_i are the injected active and reactive powers into the bus i, respectively, V_i is the voltage of the bus i, Y_{ik} is the entry i, k of the admittance matrix, and n is the number of buses. Substituting the complex voltage and admittance variables with their exponential form yields,

$$P_i + jQ_i = |V_i| \sum_{k=1}^{n} |V_k||Y_{ik}|e^{j(\delta_i - \delta_k - \theta_{ik})} \quad (3)$$

where δ and θ are the voltage and load angles, respectively. Moreover, the entry Y_{ik} in Eq. (3) can be decomposed into $G_{ik} + jB_{ik}$, where G is the conductance and B is the susceptance value. Expanding Eq. (2) and separating the real and imaginary parts results in the following power flow equations,

$$P_i = \sum_{k=1}^{n} |V_i||V_k|[G_{ik}\cos(\delta_i - \delta_k) + B_{ik}\sin(\delta_i - \delta_k)] \quad (4)$$

$$Q_i = \sum_{k=1}^{n} |V_i||V_k|[G_{ik}\sin(\delta_i - \delta_k) - B_{ik}\cos(\delta_i - \delta_k)] \quad (5)$$

Generally, there are four quantities associated with each bus, namely, P_i, Q_i, $|V_i|$, and δ_i. Depending on the bus type, two of these variables are known, and the remaining are unknown. Iterative methods such as Gauss-Seidel and Newton-Raphson are commonly employed to solve the above nonlinear power flow equations. In the N-R method, the nonlinear simultaneous equations are linearized based on Taylor's series around an initial guess, and all of the higher-order terms are ignored. Using N-R to linearize the power flow equations in (4) and (5) yields the
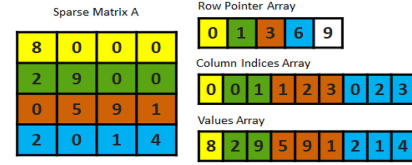

Fig. 1. A Sparse Matrix and its Corresponding CSR Arrays.

following matrix equation,

$$\begin{bmatrix} \Delta\delta \\ \Delta|V| \end{bmatrix} = J^{-1} \begin{bmatrix} \Delta P \\ \Delta Q \end{bmatrix} \quad (6)$$

where ΔP and ΔQ vectors are the mismatch between the scheduled and calculated power values (i.e., power mismatch), and the entries of the Jacobian matrix, J, are partial derivatives of Eqs. (4) and (5). As shown below, the Jacobian matrix has four varying dimension submatrices depending on the number of PQ (n_{pq}) and PV (n_{pv}) buses.

Generally, the NRPF algorithm is comprised of three main functions: a routine to build the nodal admittance matrix, an iteration loop to minimize the power mismatch, and a function to calculate network information such as line losses, branch flows, etc. The entire process is presented in Table I. Typically, control parameters are adjusted in an outer loop to correct violations in bus voltages, reactive power at generator buses, and so on. This loop adjusts system parameters, such as transformer tap and shunt control, and reruns the iteration loop to remove the violations. There is no difference in the pseudo-code of the NRPF algorithm for sparse and dense matrix implementations. However, programs that employ sparse matrix techniques are more complicated to code. For instance, one of the difficulties that we encountered was forming/updating the sparse Jacobian matrix (line 7) with the calculated values of partial derivatives (line 5). This paper presents a novel parallel solution in Section IV-D to address this problem.

IV. DESIGN AND IMPLEMENTATION

Generally, improving the execution time of a program is not a straightforward task. It involves examining computational bottlenecks, redesigning algorithms, selecting high-performance libraries, and converting code to utilize parallel processing. All of the building blocks and design schemes used in this study to maximize the peak performance of the NRPF algorithm are discussed in this section.

A. Sparse Matrix Storage

A sparse matrix is defined as a matrix with few non-zero elements. Storing only the non-zero values and their respective row
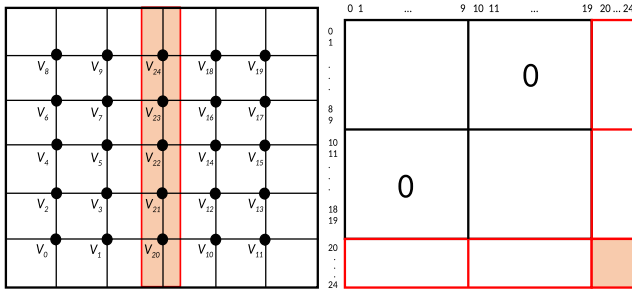
Fig. 2.   An Illustration of Nested Dissection Algorithm.

TABLE II
IMPACT OF MATRIX REORDERING ON THE NUMBER OF NONZEROS ELEMENTS

Case Name	Nonzeros in L and U (Original J)	Nonzeros in L and U (Reordered J)
Case188	13,113	182
Case300	112,134	531
Case1354pegase	2,090,398	2,448
Case2869pegase	8,793,822	5,228

and column indices will significantly reduce memory usage and computation time, because operations on zero elements are eliminated. The power flow problems have a sparse nature because power system nodes are only connected to a few adjacent buses, which makes the nodal admittance matrix highly sparse [16]. Therefore, this sparse property can be effectively utilized to improve computational performance and reduce memory usage. There are several standard formats for storing a sparse matrix. In this study, compressed sparse row (CSR) is selected for storing the sparse arrays since it is widely supported by existing libraries. Fig. 1 shows the CSR representation of matrix A. It is important to mention that external libraries were not used to produce or modify the sparse CSR arrays due to the unsatisfactory performance of those libraries.

B. Fill-Reducing Ordering of Sparse Jacobian Matrix

The N-R method for solving power flow entails finding the solution of $Ax = b$ in each iteration. The computational work and memory for solving this sparse linear system of equations depends on the *fill-in* of the factorization process. *fill-in* happens when zero values turn into non-zeros during the factorization. A standard technique to reduce the *fill-in* and improve the computational performance is to reorder the rows and columns of matrix A and solve the equivalent system $(PAQ)(Q^Tx) = Pb$, where P and Q are permutation matrices. The rearrangement process depends only on the pattern of non-zero elements in matrix A and not the values. The reordering algorithms to reduce the *fill-in* are broadly categorized into minimum degree and nested dissection methods. The first is naturally sequential but fast to compute. The latter uses a divide and conquer strategy on the graph to detect and eliminate reordering and is suited for parallel implementation.

As illustrated in Fig. 2, in nested dissection method, we partition vertices V into V_1, V_2, and V_S, where V_S is called the separator (i.e., highlighted points in red). If we first reorder V_1 and V_2, and then V_S, no *fill-in* will occur between V_1 and V_2,
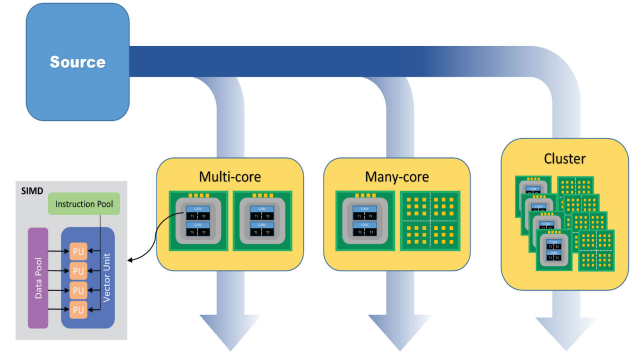

Fig. 3.   Levels of Hardware Parallelism.

and it creates a pattern of zeros similar to the top-right figure. Additionally, because the elimination process in V_1 and V_2 are independent, it can be performed in parallel. It should be noted that V_1 and V_2 can themselves be reordered by a separator to create a recursive structure in the matrix.

Because nested dissection has a parallel nature and typically results in less *fill-in* for large problems compared to other methods, we selected this algorithm for fill-reducing of the sparse Jacobian matrix. Table II shows the number of nonzero elements in the LU factor matrices of the sparse Jacobian matrix before and after applying the nested dissection algorithm. The details of these power system cases are provided in the next section.

There are several high-performance sparse direct solvers for multicore architectures such as Intel Math Kernel Library (aka Intel MKL), PETSc, MUMPS, Eigen, Pardiso, and SuiteSparse, to name a few. This study employed Intel MKL PARDISO [29] direct sparse solver because it is robust, memory-efficient, targets both SIMD and multicore parallelization, and outperformed other CPU-based libraries in most cases [7], [16], [19]. This library also supports various sparse storage formats, matrix types, and provides many numerical options. Additionally, the nested dissection algorithm is supported by this solver, which makes it easier to integrate this method into the code.

C. Multicore Processing and SIMD Vectorization

The operating frequency (i.e., clock) of computer processors was continuously increasing for many years until heat generation became an obstacle and forced the clock speed to remain in the 1-5 GHz range for about a decade now. However, chip designers overcame this problem and addressed the increasing demand for higher computational power by designing single processors, with multiple low-frequency low-voltage computing cores inside. These multicore processors can achieve high computational throughput by dividing the work among multiple resources while the power consumption and heat generation are significantly reduced. Generally, hardware parallelism can take place at different levels: Single Instruction, Multiple Data (SIMD) unit, multicore, multi-socket, many-cores, and clusters, as depicted in Fig. 3. Each of these technologies is powerful on its own, but only when combined can the highest performance be achieved. This study targets both SIMD and multicore level parallelism for solving power flow.
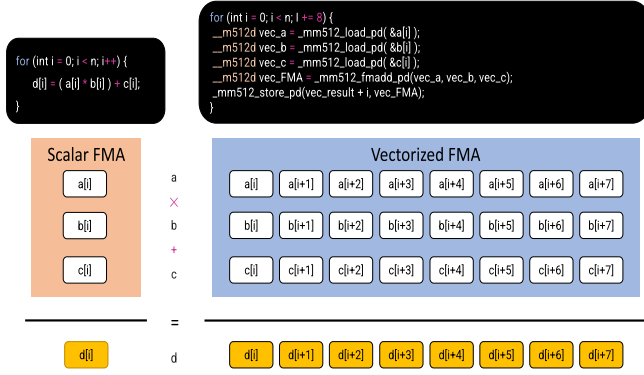
Fig. 4.   SIMD Vectorization.

Fig. 5.   Parallel OpenMP Loop.

1) Exploiting SIMD Vectorization: SIMD performs same instruction on multiple data elements in a clock cycle. Today's modern processors are equipped with up to 512 KB vector registers that offer higher data-level parallelism while using less power. For instance, a CPU with an AVX-512 instruction set can hold 8×64-bit double precision floats and execute a single instruction on them in parallel, which is 8 times faster than performing a single instruction at a time. Although modern CPUs provide direct access for SIMD level vectorization, exploiting this feature is not automatic and requires advanced programming skills and software changes. Modern compilers allow access to SIMD features by using intrinsic functions similar to the C style function calls instead of writing the code in assembly language. Fig. 4 describes the difference between scalar and vectorized SIMD operation of a Fused Multiply Add (FMA) instruction for calculating combined addition and multiplication. Data types have the following naming convention,

```
__m<register_width_in_bits><data_type>
```

where register_width_in_bits is architecture dependent (e.g., 64 bits in MMX, 128 bits in SSE, 256 bits in AVX, etc.) and data_type is either *d, i,* or no letter for double precision, integer, or single precision floating point, respectively. For example, __m256 d vector contains eight 64-bit doubles. Intrinsic functions have the following format,

```
_mm<register_width_in_bits>_<operation>_<suffix>
```

where register_width_in_bits is similar to data type, *operation* is arithmetic, logical or data operation to perform on the data stream, and *suffix* is one or two letters that indicate the data type the instruction operates upon. For example, _mm512_fmadd_pd performs fused multiply add on three given vectors as demonstrated in Fig. 4. Notice that single underscore prefix is used to call intrinsic functions and double underscore is used to declare data types. In this study, the high-performance Intel MKL functions are thoroughly used to maximize the peak performance of sparse matrix operations since they are optimized to take advantage of both SIMD and multicore level parallelism (i.e., mkl_sparse_spmm, mkl_sparse_z_add, mkl_sparse_z_mm, mkl_sparse_z_mv).

2) Shared-Memory Parallel Programming: there are a handful of extensions for developing shared-memory parallel programs, including pthreads [30], OpenMP [31], OpenACC [32], Intel TBB [33], etc. Among these extensions, OpenMP, which
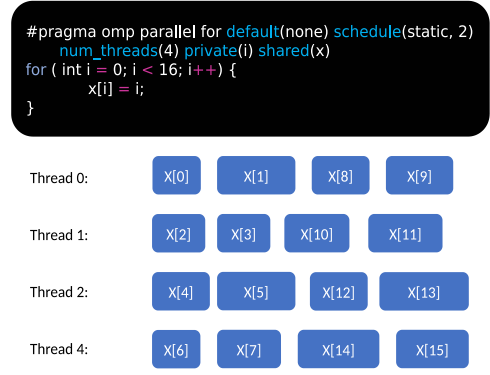
stands for Open Multi-processing, is an Application Program Interface (API) that provides a powerful directive-based programming model for developing parallel applications on platforms ranging from multicore and shared-memory systems, to coprocessors and embedded systems. OpenMP API supports parallel programming in C, C++, and Fortran languages. There are many advantages associated with OpenMP: the developed code is portable between standard compilers and operating systems, and existing sequential programs can conveniently be converted to parallel implementations. In OpenMP, the compiler handles the low-level implementation details for generating and terminating the threads. Most compilers support OpenMP.

The NRPF code is comprised of many for-loops that can effectively be parallelized with OpenMP directives. Without OpenMP, the for-loop iterations in Fig. 5 are processed in serial and by only one thread. On the other hand, the *omp parallel for* pragma provides a way to utilize multiple CPU cores and perform the calculations faster and more efficiently. This directive allows the user to specify several configuration parameters: the number of threads to use, shared and private variables, workload scheduling mechanism, and reduction operation. Workload scheduling is helpful when the amount of work varies across iterations. For example, if no scheduling mechanism is specified, all large jobs may be assigned to thread 0, and all small jobs may be processed by thread 1. Hence, thread 1 will finish faster, and will stay idle until thread 0 completes its task. As a result, the total execution time will significantly increase. OpenMP provides three scheduling options to balance the workload: static, dynamic, and guided. Interested readers are referred to the OpenMP manual to understand the difference between each work scheduling mechanism. However, based on extensive testing and evaluation, guided scheduling was determined to be the best choice in terms of performance and accuracy of the results for this study.

Another OpenMP for-loop option that was widely used in our implementation of NRPF is the parallel reduction primitives. Reductions are used in loops with dependencies, where a series of variables must be processed to produce an output (e.g., sum, max, min) at the end of the parallel region. Unlike atomic operations, thread synchronization is not needed in the case of parallel reductions. Thus, the overall performance is much
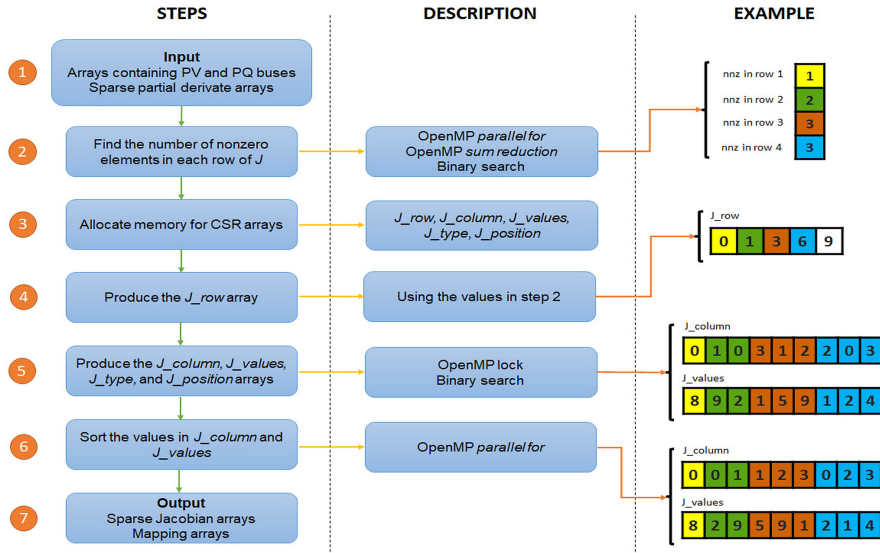
Fig. 6.   Proposed Algorithm for Forming the Sparse Jacobian Arrays.

higher. A parallel reduction can reduce the operations from N steps to $\log_2 N$ steps [19].

D. Initializing and Updating the Sparse Jacobian Matrix

Populating the Jacobian matrix in the dense implementation of NRPF is straightforward because the matrix dimension is $(n_{pv} + n_{pq} \times 2)^2$, and each entry can bed directly accessed given the row and column index. Thus, memory allocation at the beginning and updating the values at each iteration can be performed quickly. However, this is not the case for the sparse implementation because the total number of nonzero elements in the matrix and the number of nonzero entries in each row are needed to allocate memory for the CSR arrays. Therefore, two problems must be addressed: forming the CSR arrays and updating the values at each iteration.

We propose a parallel workflow for generating the CSR arrays using the OpenMP lock mechanism and the binary search algorithm. The OpenMP lock mechanism is necessary to ensure that only one thread can modify a data element at a time. We also need to determine whether a bus number belongs to a PV or PQ bus during the process. Binary search is beneficial for this purpose. It is a fast algorithm to find the index of a value within a sorted array by repeatedly splitting the search space in half and comparing the target value with the middle element of the list until the value is found or the list is empty.

Creating sparse Jacobian arrays is time-consuming, and repeating the same procedure at every iteration will undesirably affect the execution time. To address this issue, we develop two mapping arrays while populating the Jacobian CSR arrays in the first iteration and use them in successive iterations to update the Jacobian values. We take advantage of the fact that the Jacobian matrix's sparse row and column arrays remain the same, and only the values change. The first mapping array (J_{type}) refers to one of the four partial derivative arrays $J1$ to $J4$, and the second mapping array ($J_{position}$) refers to the position of that value in the respective partial derivative array. So, after the first iteration,

the program only uses these two mapping arrays to locate and update the values of the Jacobian matrix. The proposed strategy is shown in Fig. 6. Examples are provided in this figure to better illustrate the output of each step in the process. These examples relate to the matrix A in Fig. 1 and are only for illustrative purposes. The key steps are as follows:

1) The function receives the sparse sub-matrices $J1$ to $J4$ that contain the partial derivative values, together with the list of PV and PQ buses.

2) The number of nonzero elements in each row of the Jacobian matrix are calculated by processing the $J1$ to $J4$ arrays. This step is parallelized by using the OpenMP for-loop pragma and sum reduction. The number of nonzero elements is necessary to allocate memory and create the row pointer array J_{row}. The binary search is used to quickly find the bus types.

3) Next, the sparse column J_{column} and value J_{values} arrays are constructed by processing the input data. This step is parallelized with OpenMP locks so that multiple threads can work on values in each matrix row simultaneously. We also create the mapping arrays during this process.

4) Because of the parallel approach used in the previous step, the column indices in the J_{column} array are not in incremental order. We use a parallel sorting algorithm to sort the column indices in increasing order. The mapping arrays are also sorted accordingly.

To further explain the parallel approach for constructing the sparse J_{column} array, let us assume that the second row of the matrix A in Fig. 1 is being processed by two threads in parallel. There are two nonzero columns in this row. Suppose that the thread that is processing the column number of value "9" has finished first, and the column number is to be saved in the J_{column} array. This thread is aware that positions one and two in the column indices array are dedicated to the second row, but it is unaware of which position belongs to the column index of the "9" entry. Therefore, this thread begins by checking the status of position one in the J_{column} array and locks the access to it

TABLE III
SUMMARY OF TEST POWER SYSTEMS

Case File Name	Bus	Branch	Generator	PV Bus	PQ Bus
Case118	118	186	54	53	64
Case300	300	411	69	68	231
Case1354pegase	1,354	1,991	260	259	1,094
Case2383wp	2,383	2,896	327	326	2,056
Case2869pegase	2,869	4,582	510	509	2,359
Case6470rte	6,470	9,005	1,331	453	6,016
Case9241pegase	9,241	16,049	1,445	1,444	7,796
Case13659pegase	13,659	20,467	4,092	4,091	9,567
Case_ACTIVSg25k	25,000	32,230	4,834	2,752	22,247
Case_SyntheticUSA	82,000	104,121	13,419	7,739	74,258

TABLE IV
COMPARISON OF THE SOLUTION ACCURACY BETWEEN MATPOWER AND
PROPOSED IMPLEMENTATION (TOLERANCE 1E-8)

Case Name	No. of Iterations		Difference in Error Norm		
	Proposed	MATPOWER	f_0	f_{n-1}	f_n
Case118	3	3	0.0	0.0	2.7045e-14
Case300	5	5	0.0	0.0	2.8786e-15
Case1354pegase	4	4	0.0	0.0	1.7792e-12
Case2383wp	6	6	0.0	0.0	8.8457e-13
Case2869pegase	6	6	0.0	0.0	3.3957e-14
Case6470rte	3	3	0.0	0.0	3.0867e-12
Case9241pegase	6	6	0.0	0.0	2.1241e-12
Case13659pegase	5	5	0.0	0.0	1.5556e-13
Case_ACTIVSg25k	4	4	0.0	0.0	9.8761e-16
Case_SyntheticUSA	6	6	0.0	0.0	1.3699e-13

so that other threads will not be able to make any changes. If the value in this position is -1 (i.e., this array is initialized with -1), the respective column index will be written in this position, and the lock will be released. Otherwise, the thread checks the status of the next position until a vacant position is found. Due to the nature of this parallel process, the order of index values in the J_{column} array may not be incremental. Therefore, a sorting algorithm is utilized at the end to address this matter.

V. EXPERIMENTAL SETUP

A. Software and Hardware Configuration

The proposed NRPF code is developed in C++ and compiled as a 64-bit application, using the Intel compiler with /Q2 switch. We used the Intel MKL library that comes with Intel OneAPI Base Toolkit 2021.3. The *dsecnd* function of MKL library was employed to measure the time duration throughout this study. Also, the peak memory usage was measured by using the *mkl_peak_mem_usage* function. The operating system of the test machine was CentOS Linux 8.2 Core. The experiments in this study are conducted on a two-socket machine with Intel Xeon Gold 6148 processors (total of 40 cores with a base clock speed of 2.4 GHz and 376 GB of DDR4 DRAM). Each CPU core is equipped with AVX-512 instruction set and two FMA instructions. Hyperthreading was disabled during the experimental evaluation; therefore, the number of threads is equal to the number of actual CPU cores.

B. Benchmark Systems

The benchmark power systems in this study are taken from the MATPOWER 7.1 [34] package. These power systems are used to test the performance of our NRPF program and provide a comparison with other studies. Table III shows the summary information of each test case. The 118 and 300 bus systems are standard IEEE test cases. The power systems containing 1354, 2869, 9241, and 13659 buses represent different parts of the European high voltage transmission network. The power flow data for Polish and French networks are embedded in the 2,383 and 6,470 bus cases, respectively. Lastly, the synthetic 25,000 and 82,000-bus transmission systems are developed to mimic the U.S. North East and the entire continental U.S. power grid models [35]. Selected swing bus numbers in this study are similar to those in MATPOWER.

VI. EXPERIMENTAL RESULTS

A. Accuracy of the Program

The accuracy of the developed NRPF code in this study is compared with MATPOWER 7.1 as the reference. MATPOWER is a well-known, high-performance, and open-source power system simulation package based on MATLAB. Both accuracy and speed tests in MATPOWER have been performed on the above-mentioned computer using MATLAB 2021a. It is important to mention that there is no difference in the convergence property of the proposed NRPF implementation and the original NRPF method. Results in Table IV show that the number of iterations to converge to the solution is the same as MATPOWER. Additionally, the difference in the value of error norm between MATPOWER and our program for three stages of the iterative process is presented. Here, f_0 refers to the error norm difference at the beginning of the process, f_{n-1} is the difference before the last iteration, and f_n is the difference value in the final iteration. The insignificant difference in the final values of error norm is due to the conversion of input data to double-precision format and rounding in floating point. In fact, error values are exactly the same prior to the last iteration.

B. Performance Analysis

This section provides a comprehensive insight into the performance of the proposed implementation. We evaluate two scenarios and each is discussed as follows:

1) Case #1: Table V presents the average runtime obtained for essential steps of the NRPF program when both SIMD and OpenMP are utilized (i.e., the impact of matrix reordering and use of sparse arrays is not presented separately, but without applying these methods, the execution time and memory usage are many times higher than the obtained results). Note that parallelization at the SIMD level is still active when only one CPU core is utilized. The bold number in front of the function name shows the total computation time for that function. This value may be slightly higher than the sum of time values for the listed steps because some unimportant steps are omitted due to the space limit. Execution time is recorded for utilizing one and multiple CPU cores. In the case of multiple cores, the indicated core number is when the peak performance was achieved. Important observations are summarized as follows:

TABLE V
AVERAGE EXECUTION TIME OF PROPOSED NEWTON-RAPHSON POWER FLOW IMPLEMENTATION WITH SIMD AND MULTICORE LEVEL PARALLELISM (SECOND, 100 TRIALS, TOLERANCE 1E-8)

Number of Cores	Case1354pegase 1	2	Case2869pegase 1	2	Case6470rte 1	4	Case9241pegase 1	4	Case13659pegase 1	6	Case_ACTIVSg25k 1	8	Case_SyntheticUSA 1	8
Build Ybus	**0.0018**	**0.0017**	**0.0035**	**0.0027**	**0.0057**	**0.0047**	**0.0101**	**0.0073**	**0.0137**	**0.0097**	**0.0183**	**0.0119**	**0.0562**	**0.0345**
Peak Memory Usage (MB)	1.29	1.29	2.47	2.47	4.68	4.68	7.96	7.96	10.26	10.26	16.43	16.43	52.74	53.41
Power Flow Loop	**0.0501**	**0.0459**	**0.1435**	**0.1156**	**0.1141**	**0.0667**	**0.3702**	**0.268**	**0.4756**	**0.3189**	**0.6372**	**0.4431**	**3.0521**	**1.8337**
memory allocation	0.0000	0.0000	0.0000	0.0000	0.0001	0.0001	0.0002	0.0002	0.0006	0.0002	0.0007	0.0004	0.0033	0.0020
initialize V and evaluate F(V0)	0.0001	0.0001	0.0002	0.0001	0.0004	0.0002	0.0006	0.0002	0.0011	0.0003	0.0017	0.0005	0.0054	0.0013
calculate partial derivatives	0.0026	0.0029	0.0086	0.0068	0.0071	0.0048	0.0250	0.0104	0.0396	0.0125	0.0396	0.0148	0.2044	0.0789
form sparse Jacobian arrays	0.0050	0.0049	0.0122	0.0101	0.0277	0.0212	0.0459	0.0275	0.0645	0.0390	0.1041	0.0624	0.3526	0.2021
update Jacobian matrix	0.0001	0.0001	0.0005	0.0004	0.0005	0.0003	0.0020	0.0005	0.0025	0.0006	0.0036	0.0009	0.0233	0.0046
solve Ax = b	0.0417	0.0372	0.0850	0.0827	0.1062	0.0879	0.2927	0.279	0.3618	0.2645	0.4820	0.3620	2.4291	1.5388
update voltages	0.0001	0.0001	0.0006	0.0004	0.0005	0.0003	0.0014	0.0005	0.0017	0.0005	0.0024	0.0006	0.0143	0.0026
calculate injected bus powers	0.0001	0.0001	0.0004	0.0004	0.0003	0.0002	0.0029	0.0005	0.0021	0.0004	0.0021	0.0006	0.0166	0.0022
calculate norm	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0001	0.0000	0.0001	0.0000	0.0002	0.0001	0.0009	0.0002
Peak Memory Usage (MB)	25.58	24.99	36.93	44.53	55.95	86.61	82.90	167.51	86.51	177.57	133.36	279.70	347.06	548.45
Power Flow Solution	**0.0002**	**0.0003**	**0.0004**	**0.0005**	**0.0008**	**0.0007**	**0.0014**	**0.0012**	**0.0030**	**0.0017**	**0.0049**	**0.0047**	**0.0401**	**0.0087**
update generators P and Q	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0001	0.0001	0.0003	0.0002	0.0003	0.0002	0.0024	0.0011
calculate branch flows	0.0000	0.0000	0.0001	0.0001	0.0002	0.0001	0.0004	0.0003	0.0007	0.0004	0.0020	0.0021	0.0135	0.0022
calculate line losses	0.0000	0.0000	0.0000	0.0000	0.0001	0.0001	0.0001	0.0000	0.0004	0.0001	0.0007	0.0004	0.0087	0.0012
calculate line charging injections	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0001	0.0000	0.0002	0.0000	0.0003	0.0001	0.0086	0.0003
calculate misc. information	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0001	0.0001	0.0003	0.0003	0.0004	0.0003	0.0019	0.0012
Peak Memory Usage	3.68	3.68	8.77	8.77	14.03	14.03	28.71	28.71	35.71	35.71	58.58	58.58	206.49	206.49
Total Execution Time	**0.05212**	**0.04806**	**0.11214**	**0.10488**	**0.15014**	**0.12109**	**0.38181**	**0.27695**	**0.49249**	**0.33036**	**0.66054**	**0.45980**	**3.14855**	**1.87708**
Standard Deviation	**±0.001**	**±0.002**	**±0.002**	**±0.005**	**±0.008**	**±0.008**	**±0.013**	**±0.014**	**±0.020**	**±0.023**	**±0.022**	**±0.024**	**±0.053**	**±0.064**
Execution Time with Qlim Enforced	0.0933	0.0925	0.2514	0.2512	0.3752	0.3200	0.8843	0.6920	0.7473	0.5263	2.4976	1.9157	10.7185	6.7899
Total Iterations with Qlim Enforced	9	9	14	14	9	9	14	14	8	8	16	16	20	20

1) Results show that combining sparse techniques, mathematical methods, parallel processing, and existing high-performance libraries can significantly reduce the computation time of the NRPF method. Even when only one CPU core is used, SIMD level parallelism provides very satisfactory performance. It is also observed that utilizing multiple CPU cores further improved the performance. However, performance enhancement in smaller power systems is less tangible because of their smaller workload.

2) On the other hand, there is a trade-off between the number of CPU cores and peak performance achievement. As shown in Fig. 7, execution time starts to degrade after utilizing a certain number of cores due to the synchronization overhead. Our analysis shows that all parts of the algorithm begin to slow down once the saturation point is reached, but the performance degradation of computationally expensive routines is more significant than other parts.

3) Solving the linear system of equations is the most time-consuming part of the NRPF algorithm. With the re-ordering strategy, solver time is 5 to 20 times higher. On the other hand, forming the sparse Jacobian arrays is the second most time-consuming routine in the PF loop. Comparing the runtime between one and multiple CPU cores shows that our proposed parallel approach improved the speed up to $1.75\times$. More importantly, we were able to eliminate this operation in consecutive iterations by utilizing the mapping arrays that were developed when forming the sparse Jacobian arrays. For instance, the SynthicUSA case would require one additional second to form the sparse Jacobian matrix without this strategy.

4) Prior knowledge of memory usage is beneficial when it comes to simultaneous execution of multiple power flow studies or when the target processor is a GPU. Please note that in many of the Intel MKL routines, memory buffers for each thread are created. Thus, peak memory usage can be affected by the number of threads. The highest memory
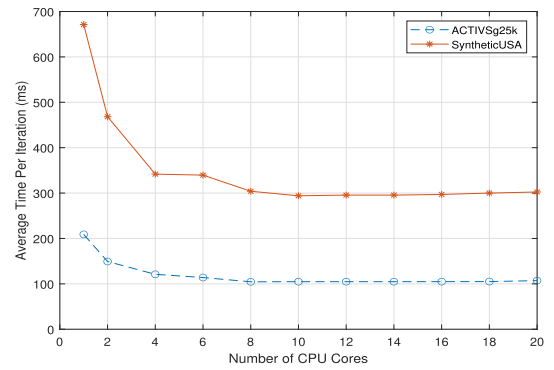

Fig. 7.   Average Iteration Time vs. Number of Cores.

usage belongs to the SyntheticUSA case with 548 MB, which means running ten concurrent power flow studies would require more than 5.5 GB of memory.

5) Enforcing the reactive power limit of generators (i.e., Qlim) results in more iterations, which ultimately leads to slower execution time. This feature is typically implemented as an outer loop around the core power flow function to correct the violations in a brute force fashion by converting the generator's bus type to PQ, turning the unit off, and re-running the power flow. Comparing one and multiple core results further confirms the importance and effectiveness of using OpenMP to thread the NRPF code on top of the SIMD parallelism, especially when equipment limits are enforced.

2) Case #2: We have so far investigated the performance of running a single power flow study with SIMD and OpenMP parallelization. In this scenario, we aim to study the impact of our proposed NRPF on the performance of applications that rely on many PF calculations. For this purpose, we have developed a contingency analysis program with the proposed NRPF as its core algorithm. Similar to Case #1, solution tolerance and the maximum number of iterations is 1E-8 and 10, respectively. The number of CPU cores is the same as those used previously to

TABLE VI
EXECUTION TIME OF CONTINGENCIES (SECOND)

Case Name	Qlim Not Enforced		Qlim Enforced	
	SIMD+Multicore	SIMD	SIMD+Multicore	SIMD
Case1354pegase	75.9	80.4	77.2	83.1
Case2869pegase	150.2	168.6	156.2	180.6
Case6470rte	390.35	450.3	562.4	693.4
Case9241pegase	562.9	712.9	625.4	802.81
Case13659pegase	1319	1770.3	1342.1	1808.4

TABLE VII
PERFORMANCE COMPARISON OF IMPLEMENTED NEWTON-RAPHSON POWER
FLOW WITH OTHER REFERENCES

Number of Buses	Reference	Platform	Runtime (Seconds)
118	[11]	FPGA-Dense	0.198
	[6]	CPU-Dense	0.019
	[18]	GPU-Sparse	0.016
	[17]	GPU-Dense	0.199
	[19]	GPU-Sparse	**0.006**
	Proposed	CPU-Sparse	0.009
	MATPOWER-NR	CPU-Sparse	0.058
	MATPOWER-NR-Qlim	CPU-Sparse	0.101
	MATPOWER-FD	CPU-Sparse	0.003
300	[11]	FPGA-Dense	2.58
	[6]	CPU-Dense	0.079
	[18]	GPU-Sparse	0.115
	[17]	GPU-Dense	2.68
	[15]	GPU-Sparse	0.95
	[19]	GPU-Sparse	0.023
	[16]	Hybrid-Sparse	0.038
	Proposed	CPU-Sparse	**0.019**
	MATPOWER-NR	CPU-Sparse	0.077
	MATPOWER-NR-Qlim	CPU-Sparse	0.141
	MATPOWER-FD	CPU-Sparse	0.006
1,354	[6]	CPU-Dense	90.77
	[16]	Hybrid-Sparse	0.077
	Proposed	CPU-Sparse	**0.048**
	MATPOWER-NR	CPU-Sparse	0.081
	MATPOWER-NR-Qlim	CPU-Sparse	0.168
	MATPOWER-FD	CPU-Sparse	0.016
2,383	[18]	GPU-Sparse	10.24
	[16]	Hybrid-Sparse	0.144
	Proposed	CPU-Sparse	**0.093**
	MATPOWER-NR	CPU-Sparse	0.126
	MATPOWER-NR-Qlim	CPU-Sparse	0.332
	MATPOWER-FD	CPU-Sparse	0.032
2,869	[16]	Hybrid-Sparse	0.221
	Proposed	CPU-Sparse	**0.104**
	MATPOWER-NR	CPU-Sparse	0.140
	MATPOWER-NR-Qlim	CPU-Sparse	0.391
	MATPOWER-FD	CPU-Sparse	0.031
8,503	[15]	GPU-Sparse	0.733
9,241	[7]	CPU-Sparse	0.288
	[13]	CPU-Sparse	197.247
	[13]	GPU-Sparse	86.46
	Proposed	CPU-Sparse	**0.276**
	MATPOWER-NR	CPU-Sparse	0.312
	MATPOWER-NR-Qlim	CPU-Sparse	1.10
	MATPOWER-FD	CPU-Sparse	0.113
23,215	[36]	GPU-Sparse	0.342
	[36]	CPU-Sparse	1.339
25,000	Proposed	CPU-Sparse	**0.459**
	MATPOWER-NR	CPU-Sparse	0.504
	MATPOWER-NR-Qlim	CPU-Sparse	3.83
	MATPOWER-FD	CPU-Sparse	0.290
82,000	Proposed	CPU-Sparse	**1.87**
	MATPOWER-NR	CPU-Sparse	3.91
	MATPOWER-NR-Qlim	CPU-Sparse	Diverged
	MATPOWER-FD	CPU-Sparse	12.1

reach the maximum peak performance. The N-1 contingency analysis in this study considers the loss of one generator or one transmission line in the system. However, we limit the line contingencies to a maximum of 2,000 to restrict the excessive calculation time for large power systems. Additionally, the test power systems are first solved to create a base case. Afterward, each contingency scenario uses a copy of this base case to apply the outage and solve the power flow.

Four combinations are simulated to tabulate the data in Table VI. When OpenMP is not active, multicore processing of for-loops plus the parallel strategy for forming Jacobian arrays is disabled. However, the SIMD level parallelization is always active regardless of the OpenMP status. Results show that using OpenMP improves the performance anywhere between 5% to 35% depending on the power system size. Typically, the performance gain is higher for larger power systems. This observation is consistent with the results in the previous scenario. It is also possible to solve multiple contingency cases concurrently. However, this study is left for future research.

C. Comparison With Other Studies

Conducting a fair performance comparison between our proposed implementation of NRPF with other studies was very challenging. Studies are different in terms of the implementation platform (i.e., CPU, GPU, Hybrid), matrix storage format (i.e., dense, sparse), functions that are included in the total execution time (i.e., sometimes only the power flow loop has been studied), and test power systems (i.e., different sizes). There are also some recent studies with a focus on concurrently running many power flows on CPUs or GPUs, but without providing the execution time for only one instance of the NRPF. To the extent possible, we attempted to provide a complete comparison between the proposed implementation and other references. Only the fastest reported runtime is selected from studies with multiple implementations. The same number of CPU cores as in Table V was used to benchmark the MATPOWER performance (i.e., two cores for 118 and 300 buses systems). MATPOWER-NR-Qlim is the runtime of MATPOWER when generator's reactive power limit was enforced.

As shown in Table VII, the only instance where GPU was able to compete with the multicore CPUs is the 118-bus system; even though the difference is less than 3 ms. For medium and large power systems, the runtime of our developed NRPF is 1.5-2.5× better than MATPOWER. For instance, the execution time of the proposed algorithm is 2.1× better than MATPOWER for the 82,000-bus system, which is also the fastest reported runtime in the literature for a system of this size at the time of publication. It is worth mentioning that MATLAB uses *memoization* and

advance caching techniques to improve the performance in the subsequent executions of a program. Because we run MATPOWER several times to obtain the average execution time, these techniques influence the reported value, and the actual runtime is expected to be slower. As expected, fast decoupled power flow is typically 2-3 times faster than the N-R method because it is the simplified version of N-R with less computational cost per iteration. However, it requires more iterations to converge than NRPF, which is the reason for the 12 s computation time

of the 82,000-bus power system. It is also known that FD power flow may diverge for cases that are solvable by the N-R method.

VII. CONCLUSION

In this paper, we presented the development of a fast Newton-Raphson power flow on multicore CPUs by combining software techniques, mathematical methods, and parallel processing. We proposed a parallel approach for developing the Jacobian CSR arrays and also a set of mapping arrays that eliminate the need for creating the Jacobian arrays in successive iterations. The accuracy of the proposed algorithm was verified by comparing the error norm with MATPOWER program. The code was precisely profiled to show the computation time for each step of the algorithm and provide a reference for future work in this area. Based on the results obtained, the performance of the proposed approach is better than GPU-based accelerated power flows. To the best of our knowledge, this is the first time in the literature that the performance of the NRPF algorithm on an 82,000-bus system was measured, and the proposed implementation was able to solve the system in 1.87 seconds, which is 2.1 times faster than the high-performance MATPOWER library. This study opens the opportunity to improve the performance of other applications that are reliant on PF computation.

REFERENCES

[1] R. C. Green, L. Wang, and M. Alam, "Applications and trends of high performance computing for electric power systems: Focusing on smart grid," *IEEE Trans. Smart Grid*, vol. 4, no. 2, pp. 922–931, Jun. 2013.

[2] R. C. Green, L. Wang, and M. Alam, "High performance computing for electric power systems: Applications and trends," in *Proc. IEEE Power Energy Soc. Gen. Meeting*, 2011, pp. 1–8.

[3] D. M. Falcão, "High performance computing in power system applications," in *Proc. Int. Conf. Vector Parallel Process.*, 1996, pp. 1–23.

[4] M. Čepin, *Assessment of Power System Reliability: Methods and Applications*. Berlin, Germany: Springer, 2011, pp. 147–154.

[5] H. Dağ and G. Soykan, "Power flow using thread programming," in *Proc. IEEE Trondheim PowerTech*, 2011, pp. 1–5.

[6] A. Ahmadi, S. Jin, M. C. Smith, E. R. Collins, and A. Goudarzi, "Parallel power flow based on OpenMP," in *Proc. North Amer. Power Symp.*, 2018, pp. 1–6.

[7] G. Guerra and J. A. Martinez-Velasco, "Evaluation of MATPOWER and OpenDSS load flow calculations in power systems using parallel computing," *IET Eng.*, vol. 2017, no. 6, pp. 195–204, 2017.

[8] R. D. Zimmerman, C. E. Murillo-Sánchez, and R. J. Thomas, "MATPOWER: Steady-state operations, planning, and analysis tools for power systems research and education," *IEEE Trans. Power Syst.*, vol. 26, no. 1, pp. 12–19, Feb. 2011.

[9] R. S. Kumar and E. Chandrasekharan, "A parallel distributed computing framework for Newton-Raphson load flow analysis of large interconnected power systems," *Int. J. Elect. Power Energy Syst.*, vol. 73, pp. 1–6, 2015.

[10] L. Ao, B. Cheng, and F. Li, "Research of power flow parallel computing based on MPI and P-Q decomposition method," in *Proc. Int. Conf. Elect. Control Eng.*, 2010, pp. 2925–2928.

[11] X. Wang, S. G. Ziavras, C. Nwankpa, J. Johnson, and P. Nagvajara, "Parallel solution of Newton's power flow equations on configurable chips," *Int. J. Elect. Power Energy Syst.*, vol. 29, no. 5, pp. 422–431, 2007.

[12] G. Zhou *et al.*, "GPU-accelerated batch-ACPF solution for N-1 static security analysis," *IEEE Trans. Smart Grid*, vol. 8, no. 3, pp. 1406–1416, May 2017.

[13] I. Araújo, V. Tadaiesky, D. Cardoso, Y. Fukuyama, and Á. Santana, "Simultaneous parallel power flow calculations using hybrid CPU-GPU approach," *Int. J. Elect. Power Energy Syst.*, vol. 105, pp. 229–236, 2019.

[14] S. Huang and V. Dinavahi, "Real-time contingency analysis on massively parallel architectures with compensation method," *IEEE Access*, vol. 6, pp. 44 519–44530, 2018, doi: 10.1109/ACCESS.2018.2864757.

[15] G. Zhou *et al.*, "A novel GPU-accelerated strategy for contingency screening of static security analysis," *Int. J. Elect. Power Energy Syst.*, vol. 83, pp. 33–39, 2016.

[16] X. Su, C. He, T. Liu, and L. Wu, "Full parallel power flow solution: A GPU-CPU-based vectorization parallelization and sparse techniques for Newton-Raphson implementation," *IEEE Trans. Smart Grid*, vol. 11, no. 3, pp. 1833–1844, May 2020.

[17] C. Guo, B. Jiang, H. Yuan, Z. Yang, L. Wang, and S. Ren, "Performance comparisons of parallel power flow solvers on GPU system," in *Proc. IEEE Int. Conf. Embedded Real-Time Comput. Syst. Appl.*, 2012, pp. 232–239.

[18] J. Singh and I. Aruni, "Accelerating power flow studies on graphics processing unit," in *Proc. Annu. IEEE India Conf.*, 2010, pp. 1–5.

[19] V. Roberge, M. Tarbouchi, and F. Okou, "Parallel power flow on graphics processing units for concurrent evaluation of many networks," *IEEE Trans. Smart Grid*, vol. 8, no. 4, pp. 1639–1648, Jul. 2017.

[20] C. Vilacha, J. Moreira, E. Miguez, and A. F. Otero, "Massive Jacobi power flow based on SIMD-processor," in *Proc. 10th Int. Conf. Environ. Elect. Eng.*, 2011, pp. 1–4.

[21] A. Ahmadi, F. Manganiello, A. Khademi, and M. C. Smith, "A parallel Jacobi-embedded Gauss-Seidel method," *IEEE Trans. Parallel Distrib. Syst.*, vol. 32, no. 6, pp. 1452–1464, Jun. 2021.

[22] X. Li, F. Li, H. Yuan, H. Cui, and Q. Hu, "GPU-based fast decoupled power flow with preconditioned iterative solver and inexact Newton method," *IEEE Trans. Power Syst.*, vol. 32, no. 4, pp. 2695–2703, Jul. 2017.

[23] S. Huang and V. Dinavahi, "Performance analysis of GPU-accelerated fast decoupled power flow using direct linear solver," in *Proc. IEEE Elect. Power Energy Conf.*, 2017, pp. 1–6.

[24] M. M. A. Abdelaziz, "OpenCL-accelerated probabilistic power flow for active distribution networks," *IEEE Trans. Sustain. Energy*, vol. 9, no. 3, pp. 1255–1264, Jul. 2018.

[25] D. Ablakovic, I. Dzafic, and S. Kecici, "Parallelization of radial three-phase distribution power flow using GPU," in *Proc. 3rd IEEE PES Innov. Smart Grid Technol. Europe*, 2012, pp. 1–7.

[26] M. Abdelaziz, "GPU-OpenCL accelerated probabilistic power flow analysis using Monte-Carlo simulation," *Electric Power Syst. Res.*, vol. 147, pp. 70–72, 2017.

[27] T. Cui and F. Franchetti, "A multi-core high performance computing framework for distribution power flow," in *Proc. North Amer. Power Symp.*, 2011, pp. 1–5.

[28] Intel Math Kernel Library. [Online]. Available: https://software.intel.com/mkl

[29] B. Nichols, D. Buttlar, and J. P. Farrell, *Pthreads Programming*. Sebastopol, CA, USA: O'Reilly Associates, 1996, pp. 1–29.

[30] OpenMP. [Online]. Available: https://www.openmp.org/

[31] OpenACC. [Online]. Available: https://www.openacc.org/

[32] *Intel Threading Building Blocks*. [Online]. Available: https://software.intel.com/tbb

[33] Matpower 7.0. [Online]. Available: https://matpower.org/download/

[34] A. B. Birchfield, T. Xu, K. M. Gegner, K. S. Shetye, and T. J. Overbye, "Grid structural characteristics as validation criteria for synthetic networks," *IEEE Trans. Power Syst.*, vol. 32, no. 4, pp. 3258–3265, Jul. 2017.

[35] H. Jiang, D. Chen, Y. Li, and R. Zheng, "A fine-grained parallel power flow method for large scale grid based on lightweight GPU threads," in *Proc. IEEE 22nd Int. Conf. Parallel Distrib. Syst.*, 2016, pp. 785–790.

Afshin Ahmadi (Member, IEEE) received the M.S. degree in electrical engineering from the University of the Philippines, Philippines, in 2012, and the Ph.D. degree in computer engineering from Clemson University, SC, United States, in 2020. He is currently a Power System Application Developer with the Electric Reliability Council of Texas (ERCOT). His main research interests include high-performance computing in power and energy systems, smart grid, and power system planning.

Melissa Crawley Smith (Senior Member, IEEE) received the B.S. and M.S. degrees in electrical engineering from Florida State University, Tallahassee, Florida, in 1993 and 1994, respectively, and the Ph.D. degree in electrical engineering from the University of Tennessee, TN, United States, in 2003. She is currently an Associate Professor of electrical and computer engineering with Clemson University. She has more than 25 years of experience developing and implementing scientific workloads and machine learning applications across multiple domains, including 12 years as a research associate at Oak Ridge National Laboratory. Her current research interests include the performance analysis and optimization of emerging heterogeneous computing architectures (GPGPU- and FPGA-based systems) for various application domains including machine learning, high-performance or real-time embedded applications, and image processing.

Vahid Dargahi (Member, IEEE) received the Ph.D. degree in electrical engineering, with an emphasis in power electronics and power systems, from Clemson University, Clemson, SC, USA, in 2017. From 2016 to 2017, he was a Graduate Research Assistant with the eGRID Center, CURI, North Charleston, SC, USA. From 2018 to 2019, he was a Postdoctoral Research Fellow with the Electrical and Computer Engineering Department, UC Santa Cruz, CA, USA. He is currently an Assistant Professor with the School of Engineering and Technology, University of Washington, Tacoma, WA, USA. His current research interests include power systems, parallel processing, power electronics circuits, novel converter topologies, widebandgap semiconductor devices, multilevel inverters, control of power electronic systems, grid-tied inverters, and active rectifiers.

Edward Randolph Collins (Senior Member, IEEE) received the B.S.E.E. degree from North Carolina State University, Raleigh, NC, USA, and the Ph.D. degree in electrical engineering from the Georgia Institute of Technology, Atlanta, GA, USA. In 1989, he joined the Department of Electrical and Computer Engineering, Clemson University, North Charleston, SC, where he is currently a Professor. He directs the research activities in the Power Quality and Industrial Applications Laboratory at Clemson University and has focused his research on power quality and grid integration, in addition to power electronics, machines and controls. He was in several leadership positions at Clemson related to electric power and energy, including executive Director of the Energy Innovation Center.

Shuangshuang Jin (Senior Member, IEEE) received the Ph.D. degree in computer science from Washington State University in 2007. She is an Associate Professor with the School of Computing with a joint appointment in the Department of Electrical and Computer Engineering at Clemson University. Her research interests include high-performance computing (HPC), big data analysis, and scientific computation and visualization. She has over 15 years of interdisciplinary research experience applying advanced computing technology to pressing scientific and engineering domain areas. Prior to joining Clemson University, Dr. Jin was a senior research scientist at Pacific Northwest National Laboratory, and contributed to the development of a variety of HPC-based power system applications such as look-ahead dynamic simulation, distributed state estimation, massive contingency analysis, GridLAB-D, and GridPACK, etc.